Secure Autonomous Cyber-Physical Systems Through Verifiable Information Flow Control

Jed Liu[*]
Barefoot Networks
Ithaca, NY, USA
liujed@cs.cornell.edu

Joe Corbett-Davies
Cornell University
Ithaca, NY, USA
jwc292@cornell.edu

Andrew Ferraiuolo
Cornell University
Ithaca, NY, USA
af433@cornell.edu

Alexander Ivanov
Cornell University
Ithaca, NY, USA
aii4@cornell.edu

Mulong Luo
Cornell University
Ithaca, NY, USA
ml2558@cornell.edu

G. Edward Suh
Cornell University
Ithaca, NY, USA
suh@csl.cornell.edu

Andrew C. Myers
Cornell University
Ithaca, NY, USA
andru@cs.cornell.edu

Mark Campbell
Cornell University
Ithaca, NY, USA
mc288@cornell.edu

ABSTRACT

Modern cyber-physical systems are complex networked computing systems that electronically control physical systems. Autonomous road vehicles are an important and increasingly ubiquitous instance. Unfortunately, their increasing complexity often leads to security vulnerabilities. Network connectivity exposes these vulnerable systems to remote software attacks that can result in real-world physical damage, including vehicle crashes and loss of control authority.

We introduce an integrated architecture to provide provable security and safety assurance for cyber-physical systems by ensuring that safety-critical operations and control cannot be unintentionally affected by potentially malicious parts of the system. Fine-grained information flow control is used to design both hardware and software, determining how low-integrity information can affect high-integrity control decisions. This security assurance is used to improve end-to-end security across the entire cyber-physical system. We demonstrate this integrated approach by developing a mobile robotic testbed modeling a self-driving system and testing it with a malicious attack.

ACM Reference Format:
Jed Liu, Joe Corbett-Davies, Andrew Ferraiuolo, Alexander Ivanov, Mulong Luo, G. Edward Suh, Andrew C. Myers, and Mark Campbell. 2018. Secure Autonomous Cyber-Physical Systems Through Verifiable Information Flow Control. In *Workshop on Cyber-Physical Systems Security & Privacy (CPS-SPC '18), October 19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3264888.3264889

1 INTRODUCTION

The modern world is increasingly populated by networked devices with high-performance computing, complex sensors, and significant physical actuation components [44]. This is especially true of automotive systems, which increasingly include not only Internet connectivity, but also autonomous driving capabilities. Network connectivity brings convenience, such as entertainment, maps, and traffic reports. But it can also bring remote attacks. Unlike traditional computing systems, attacks on safety-critical cyber-physical systems can also result in real-world physical damage. Vehicles and other systems can cause significant physical harm if they are maliciously compromised, so it is becoming more important to secure cyber-physical systems.

The new capabilities in networked cyber-physical systems demand more complex infrastructure and algorithms, and often lead to new security flaws [51]. For example, recent studies of modern automobiles have revealed attack surfaces across infotainment, wireless Internet, and diagnostics components, allowing an attacker to remotely control steering, brake, and throttle [11, 35]. With the widespread development and increased deployment of driver assistance and self-driving features [9], we can soon expect greater vehicle connectivity, more sensing modalities, a larger volume of data from external sources, and a higher degree of automated actuator control. Similar security concerns exist for other types of safety-critical cyber-physical systems. For instance, in Boeing's 787 Dreamliner aircraft, the avionics systems are on the same network as passenger Wi-Fi networks [65].

We propose a new approach to building cyber-physical systems with high security assurance, through integrated co-development of three layers of the system—hardware, software, and control algorithms. Because security is a system-level property, traditional single-layer protection is not enough. Even systems with perfectly secure hardware and software may be compromised if control algorithms cannot properly handle malicious inputs [41]. Similarly, even perfectly secure algorithms can be compromised if there exist vulnerabilities in underlying hardware or software layers. We incorporate protection mechanisms for hardware, software, and control algorithms to instantiate a novel system architecture for secure autonomous vehicles.

A key technical ingredient is the use of *security-typed* programming languages to develop different system layers securely, including both software and hardware. We build on security-typed languages that enforce information-flow policies at compile time using type annotations [48]. Languages such as Jif [38] and SPARK [6] have been used to enforce separation of information with differing

[*]Work done while at Cornell

confidentiality and integrity levels. These languages are highly robust to many classes of attacks commonly used to take control of cyber-physical systems. Similarly, modern hardware design techniques can produce processor architectures that prevent or limit any interference between one (possibly untrusted) process and another running on the same processor [16]. Combining these kinds of modern information security techniques at the system-design level is critical for the safety of cyber-physical systems.

We have implemented a prototype of the proposed protection architecture on a mobile robotic platform that emulates lane-following behavior by traversing a track. As the robot moves, it downloads external map information, as is prevalent in modern self-driving cars [34]. Experimental results are presented from a simulated remote attack, in which an untrusted map server on the Internet has been compromised. To defend against the attack, a map-verification algorithm verifies the map data before it is used by the robot's path planner. The planner and map verifier are developed in Jif, which enforces *information flow control* at the language level, thereby ensuring that only verified map data can reach the planner. We have also designed and verified a secure processor for providing strong isolation assurance for the planner, map verifier, and other safety-critical software in the system. The processor is not yet integrated into the prototype, but evaluation of the processor suggests an overall performance overhead for the strong isolation, including removal of timing interference, can be low: about 12%.

The prototype demonstrates that the integrated protection architecture is indeed viable in practice.Analysis of the architecture also suggests that the integrated protection approach can handle a wide range of vulnerabilities across algorithmic, software, and hardware layers. In fact, the threat analysis shows that the vertically-integrated protection across the layers is necessary in order to protect a system from a wide range of security attacks.

The rest of the paper is organized as follows. Section 2 discusses our threat model and defense strategies. Section 3 introduces an integrated system architecture for secure autonomous CPS, which include protection in algorithmic, software, and hardware layers. Section 4 describes our prototype of the proposed system architecture on a Segway-based robotic platform and our experiences with the prototype platform. Section 5 evaluates the effectiveness and the overhead of the proposed protection approaches. Section 6 discussed related work, and Section 7 concludes the paper.

2 THREAT MODEL

To make the security problems concrete, we focus on attacks that can arise in the context of autonomous vehicles, where we presume that the attacker's goal is to cause unsafe vehicle behavior by influencing safety-critical control decisions through untrusted inputs. Figure 1 summarizes potential security threats to the safe operation of an autonomous vehicle, and outlines our defense strategies for those threats that we address. Our goal is to defend the vehicle against remote adversaries who lack direct hardware access, but who can communicate with the vehicle over the network and can affect a subset of local sensor inputs through spoofing or environmental tampering.

We assume that the adversary can control untrusted network inputs, which the vehicle might use for safety-critical decisions.

Threat description	Covered?	Defense strategy
Physical attacks on vehicle		
Directly tamper with vehicle hardware	No	–
Remote software attacks through the network		
Maliciously modify network inputs to safety-critical controller	Yes	Input verification using local sensors (§3.1)
Exploit user-level software bugs that allow untrusted inputs to unintentionally affect safety-critical decisions	Yes	Memory-safe language with language-based information flow control (§3.2)
Slow down safety-critical software through hardware-level interference	Yes	Timing interference control in hardware (§3.3)
Exploit OS-level bugs to break software isolation	Yes	Formally verified microkernel OS ([26])
Exploit hardware bugs to break software isolation	Yes	Formally verified hardware information flow (§3.3)
Sensor and environmental attacks		
Maliciously affect a subset of sensor inputs	Yes	Validation using multiple sensors and inputs (§3.1)
Maliciously affect all sensor inputs together	No	–

Figure 1: Summary of threats and defenses.

For example, if the vehicle downloads maps from the Internet, the adversary might try to influence the vehicle's route by providing incorrect maps.

The adversary can also provide malicious network inputs to trigger bugs in software or hardware. For example, the adversary can provide malformed map data to try to exploit vulnerabilities in the vehicle's parsing (or other handling) of that data in order to have software mistakenly use untrusted inputs in making safety-critical decisions. For critical, trusted software, our system uses language-level information flow control to prevent such attacks.

The adversary may take over a less-trusted, user-level software component, such as an infotainment system. To model this kind of attack, we assume that the adversary already controls all untrusted software components on the vehicle. The adversary might use this control to interfere with the operation of the vehicle's trusted software components through a hardware-level denial-of-service (DoS) attack. For instance, the attacker might try to slow down critical control tasks via contention for hardware resources such as caches, on-chip interconnects, and memory (DRAM) channels. In fact, maliciously creating heavy memory traffic on a multi-core processor can cause other programs to slow down by an order of magnitude [37]. Such timing attacks can cause delays in crucial data processing, which may have crippling effects on control and estimation algorithms, thereby compromising the safe operation of a vehicle. For example, a collision-avoidance mechanism will not be useful if its response is delayed and triggered after a collision.

After taking control of untrusted software on the vehicle, the adversary may also try to exploit bugs in an operating system or hardware in order to influence trusted software. Software vulnerabilities in a complex operating system have been commonly exploited in traditional software attacks. To prevent such attacks, we assume that the operating system on the vehicle is hardened to ensure strong separation among software components (e.g., with a formally verified microkernel [26]).

Recent attacks such as Meltdown [30] and Spectre [27] suggest that hardware-level vulnerabilities can also be exploited to break system security. Processor errata often include security bugs [23], and there have been vulnerabilities in implementations of Intel VT-d and system management mode (SMM) [58, 59]. Vulnerabilities are also found in safety-critical hardware, including an exploitable bug in the Actel ProASIC3 [52], which has been used in medical, automotive, aerospace, and military applications, including the Boeing 787 Dreamliner aircraft. By formally verifying information flow in hardware, we can prevent hardware-level bugs from violating software isolation.

Finally, we assume that the adversary can manipulate sensor inputs (e.g., GPS, LIDAR, cameras) to attempt to influence the vehicle's control algorithms. For example, the adversary might spoof GPS signals to cause the vehicle's navigation system to report its location inaccurately, or erect billboards along the route that defeat the vehicle's vision algorithms. In general, we assume that the adversary does not have total control over all sensors and inputs, as no defense is possible against an adversary so powerful that it can place the vehicle in a self-consistent virtual world [15]. The following are some examples of attacks on sensors and inputs that we defend against.

- In a *map poisoning attack* [41], a vehicle is served a maliciously modified map by an untrusted map server on the network. In the absence of our security mechanisms, this map could cause the vehicle to navigate to the wrong destination or off the road [25]. One way to defend against this attack is to use GPS and vision, or other scene-sensor information.
- GPS spoofing might cause the vehicle to drive off the road [64]. Map data and vision, or other scene-sensor information can be used to defend against this attack.
- Radar, LIDAR, and camera inputs might be blinded or spoofed to cause the vehicle to ignore obstacles, resulting in collisions [42]. We assume that at least one obstacle detection system is not compromised by the attacker.

3 SYSTEM ARCHITECTURE

We introduce a general system architecture, illustrated in Figure 2, for securing autonomous cyber-physical systems. Sensors and external information form the external attack surface for the threats described previously. Therefore, this input to the system is, in general, a mix of trusted and untrusted information, represented by the multicolored arrows in the figure. Our approach integrates three key features to secure these autonomous cyber-physical systems:

- *Input verification, collision-avoidance software.* Sensors and other inputs are cross-validated in the Verification block of Figure 2 to detect adversarial manipulation. If an attack is detected, untrusted information is prevented from reaching subsequent software blocks. Future work will generalize this concept to define a label about how much the passed information is untrusted, which would aid the other blocks in reasoning about the incoming information. The Path/Control block is designed to use trusted information to plan a collision-free path to ensure the safe operation of the vehicle.
- *Software-level information flow control and verification.* Sensor verification, path planning, collision avoidance, and other safety-critical software components are implemented in a memory-safe, security-typed programming language to protect their integrity. Language-level information flow control ensures that untrusted sensors and inputs cannot be used for safety-critical operations without explicit verification and endorsement.
- *Verified hardware platform for information flow control.* All trusted, safety-critical software components are executed on a secure processor that we have developed. The processor controls information flows, including timing interference, and is formally verified with a hardware-level security type system co-developed with the processor. By executing trusted software components on the secure processor, we protect these components from interference by less-trusted software that may be under the control of the adversary. The processor also supports new instructions that enable the software level to communicate information flow policies to the hardware level for control of information flows across the software–hardware boundary.

Implementing a cyber-physical system with this architecture improves security, but adds costs for development effort and run-time performance. These costs are explored in Section 5.

3.1 Sensor verification and safety analysis

Autonomous vehicles require many sources of information, ranging from raw sensors to external networked services such as map servers. As shown in Figure 2, our architecture includes a verification step for all of these inputs, and prevents unverified inputs from tainting the operation of the system.

Our approach is to design the system with redundant inputs across multiple sensor modalities, and to leverage this redundancy to verify each input against the others. For example, to verify a vision detector of objects such as other cars, LIDAR and radar measurements can be used. Information identified as being corrupted is stopped at the verification step, and is prevented from propagating further in the software pipeline. Instead, a signal is sent that indicates what information is corrupted. These signals are then used to activate specific safe protocols in subsequent sensor blocks. For example, the Tactical planner requires the use of map data. But if the map data is corrupted, this fact is passed to the Tactical planner, which in turn activates a standard safe plan requiring no map data, such as safely pulling off to the side of the road.

The general question of how to detect and compensate for compromised sensors and maps in a computationally efficient manner is still open to research. However, statistical analysis seems to be a key component, because real-world environments and sensors are inherently stochastic: spurious sensor measurements occur even in

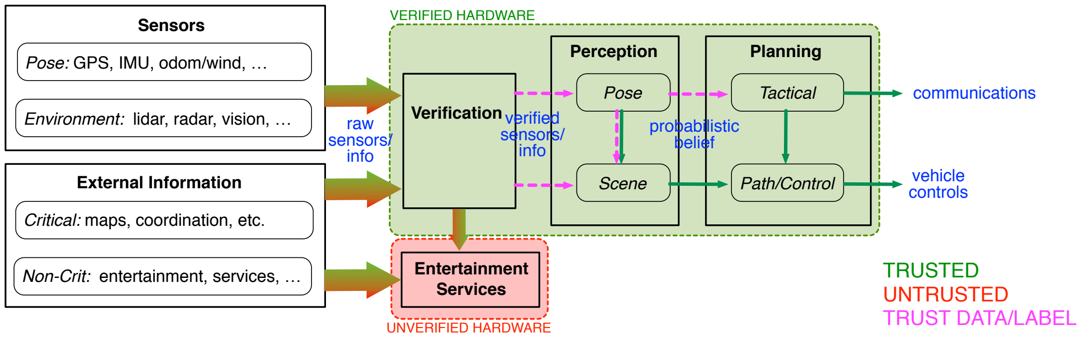

Figure 2: Autonomous CPS system architecture, with trusted and untrusted components, trust labels, and verified hardware.

the absence of an attacker. Hypothesis testing [61] is a statistical analysis primarily used to identify spurious sensor measurements rather than identifying malicious attacks. Some recent work [40] has analyzed the methods and conditions under which malicious sensors can be detected. These approaches use optimization to locate potentially malicious sensors, while assuming stochastic noise is bounded. This optimization problem can become intractable because the number of possible failure scenarios is exponential in the number of sensors. Instead, we exploit statistical properties to perform multivariate hypothesis testing.

In this work, a key question we ask is whether a piece of information $\boldsymbol{y}$, such as a measurement, is consistent with known (or expected) information $\hat{\boldsymbol{y}}$. More formally, the null hypothesis is defined as $\boldsymbol{y}$ matching a Gaussian model, or

$$H_0 : \boldsymbol{y} \sim \mathcal{N}(\hat{\boldsymbol{y}}, \Sigma)$$

where Σ is the covariance. In this case, the null hypothesis implies that a particular sensor or map is trustworthy, since a robot cannot begin to act if all of its information starts off as untrusted. The alternate hypothesis is the case where $\boldsymbol{y}$ does not match our Gaussian model. We model the alternate hypothesis via an uninformative uniform distribution:[1]

$$H_1 : \boldsymbol{y} \sim \mathcal{U}$$

The likelihood ratio is then used to define a test statistic:

$$\Lambda = \frac{p(\boldsymbol{y}|H_1)}{p(\boldsymbol{y}|H_0)} > \Lambda_0 \qquad (1)$$

Given the above likelihoods, this can be simplified:

$$p\left((\boldsymbol{y} - \hat{\boldsymbol{y}})^T \Sigma^{-1} (\boldsymbol{y} - \hat{\boldsymbol{y}}) | H_0\right) \sim \chi^2(n_y) \qquad (2)$$

The test statistic is a χ^2 random variable with n_y degrees of freedom, and can be used for statistically rigorous evaluations. For example, we might classify $\boldsymbol{y}$ as untrusted when there is a less than 2% chance that $\boldsymbol{y}$ is consistent with the other information $\hat{\boldsymbol{y}}$.

This hypothesis-testing approach can be viewed as rigorous anomaly rejection. Whereas in machine learning, the confidence of an estimate depends on the training data and may have no rigorous probabilistic meaning, hypothesis testing offers added structure that allows for a rigorous interpretation of the likelihood of a false positive or a false negative. Current research focuses on machine learning concepts that capture uncertainty rigorously, and a future research topic is to integrate these into our framework.

[1] Non-uniform distributions could be used to capture more complex a priori attacks; this is an area of future work.

3.2 Software-level information flow control

To help prevent vulnerabilities in their implementation, we write the path planner, collision avoidance, and other safety-critical functions in Jif [38], a security-typed language based on Java. Jif inherits memory safety from Java, eliminating a large class of low-level software vulnerabilities, such as buffer overruns and dangling pointers. Not only is this important for protecting the execution of Jif programs, it also prevents subversion of Jif's own security mechanisms.

Additionally, Jif enforces information flow security. Jif programs contain annotations (*labels*) that express policies for confidentiality and integrity. Security is enforced both at compile time and run time: the compiler checks that every flow of information through a program either statically conforms to the labels in the program, or has a run-time check proving that the flow is secure.

Labels are part of types in Jif. The label in the type of a variable (or expression) restricts the information that may affect the value of the variable. When flow from label L_1 to label L_2 is secure, we say that L_1 *flows to* L_2 and write $L_1 \sqsubseteq L_2$. Information flows in the program are secure if they respect this information-flow ordering. As part of type-checking programs, the Jif compiler checks all information flows accordingly. For example, an assignment of the form $x = y$ causes the compiler to check that $L(y) \sqsubseteq L(x)$, where $L(x)$ is the label of the variable x.

Jif also supports a notion of *downgrading*, in which sufficiently trusted code can relax the usual rules for information flow. For example, when the map is verified, run-time language-level *endorsement* [62] is used in our prototype to allow the map to be considered by the secure planner. Jif's enforcement ensures that this endorsement is the only way for untrusted information to affect trusted computations. Endorsements in Jif are readily identified by the `endorse` keyword, which facilitates security audits of Jif programs.

3.3 Verified hardware for software isolation

In order to ensure that processor hardware securely isolates safety-critical software from untrusted software, we augment a modern processor architecture with mechanisms for strong control of hardware-level information flow including timing interference, and verify the processor implementation using static information flow analysis at the hardware description language (HDL) level. For this hardware-level protection, we add hardware tags to the processor that explicitly indicate the current security domain of a processor core, each register, and a physical memory page. The security tag is also attached to each memory access so that timing interference in a shared memory hierarchy can be removed. The instruction set architecture (ISA) is augmented to let trusted software (such as a microkernel) manage the hardware security tags. We implemented the tagged architecture as an extension to the RISC-V Rocket processor [5], which supports a complete RISC-V ISA with all the necessary features to run an operating system. The details of the secure processor design can be found in a separate paper [17].

To remove microarchitecture timing interference, we statically allocate resources to different security domains so that the execution time of software running in one security domain is not affected by software in another security domain. We refer to security domains with timing isolation as *timing compartments*. *Spatial partitioning* removes contention by duplicating or partitioning a resource for each security domain. For example, shared caches can be partitioned so that only a subset of cache ways can be used by each security domain. *Temporal partitioning* removes contention by time-multiplexing resources among security domains, with a fixed schedule. For example, on-chip interconnect and DRAM memory channels use the fixed time-multiplexing so that memory accesses from one security domain is not affected by accesses from other domains. Static resource allocation completely removes timing interference, and makes a processor design simple to analyze for security verification.

For verification, we used the security type system from SecVerilog [67], a secure design language that extends Verilog (a popular HDL) with a security type system. SecVerilog has syntax for annotating variables (wires and registers) with security labels, and allows hardware designers to formally verify information flow security properties of their designs. This verification is timing-sensitive and ensures that timing interference among security domains is removed in the processor implementation. The Rocket processor is implemented in Chisel. To check our modified design, we ported SecVerilog's security type system to Chisel, and formally verified that our prototype processor enforces strict, timing-sensitive noninterference.

4 IMPLEMENTATION PROTOTYPE

To evaluate the feasibility and effectiveness of the proposed architecture, we implemented a core piece of an autonomous vehicle system, and used it to drive a Segway-based robotic platform to produce lane-following behavior on a single-lane track. Our aim was to understand the benefits and challenges of an integrated approach of combining algorithms, hardware, and software, when building a secure autonomous cyber-physical system.

4.1 Prototype system architecture

Figure 3 shows the core component of the prototype's architecture, a specific instantiation of the general architecture we proposed in Figure 2. The system receives map data from an untrusted map server on the Internet. A map-verification module verifies the untrusted map using trusted sensor information. If the map is successfully verified, it is explicitly endorsed for use in the path planning computation. If verification fails, the untrusted map cannot be used in trusted computations, and the path planner falls back to a plan generated exclusively from (trusted) sensor data.

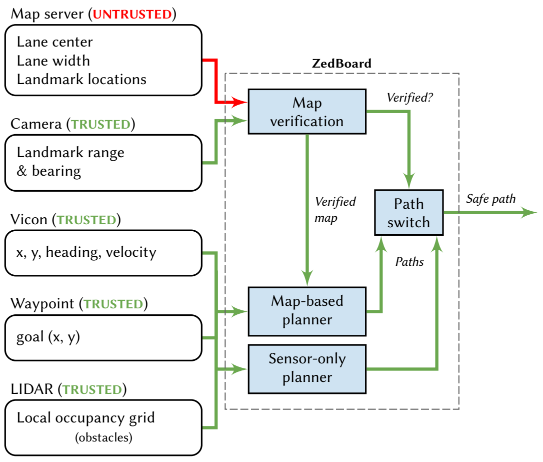

Figure 3: Architecture of the prototype system. Green and red arrows represent trusted and untrusted information flows, respectively.

The map-verification and path-planner software are written in Jif and run on a ZedBoard [63], a hardware development board based on the Xilinx Zynq-7000 SoC. Currently, the Jif code executes in a JVM running on the ZedBoard's ARM Cortex-A9 processor.

An FPGA on the ZedBoard allows the implementation of our secure processor on the platform. We implemented the verified processor as an extension to the RISC-V Rocket processor [5], which includes a pipelined in-order core with branch and branch-target prediction, an ALU, a multi-cycle multiplier, and a floating-point unit (FPU). The processor also includes 16-KB L1 instruction and data caches, and instruction and data TLBs for virtual memory support. The processor is verified with the security type system. The processor, however, is not yet used in the integrated vehicle platform: we are still working on a Jif compiler that targets the chip. The primary role of the verified hardware is to provide strong isolation assurance and does not change the functionality of the platform. The overhead of the hardware protection is evaluated separately in Section 5.3.

4.2 Sensors and platform hardware

The ZedBoard controls a robotic platform consisting of a differential-drive Segway RMP50 outfitted with sensors and processing, shown in Figure 4. On board the robot is the *Segway PC*, a dedicated computing platform that preprocesses raw sensor data for the ZedBoard, and performs path following on the ZedBoard's output. In a production implementation, this extra processing would be colocated with map verification and path planning on a secure hardware module. The ZedBoard in our prototype has limited processing capacity,

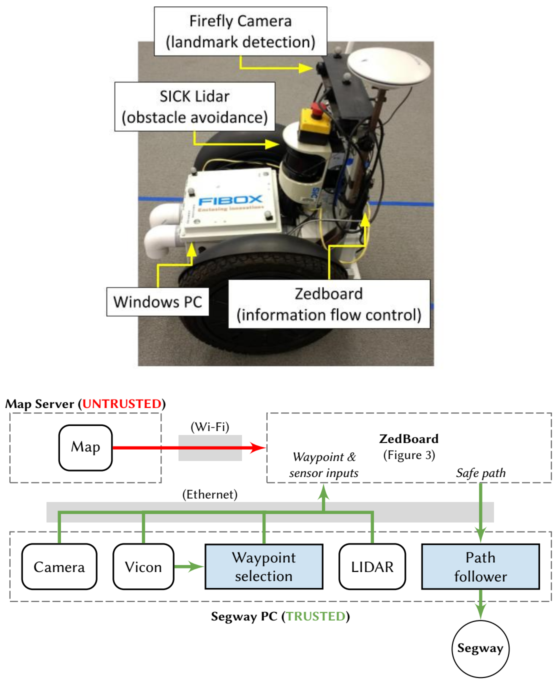

Figure 4: Segway robotic testbed. Dashed boxes indicate separate physical computing units.

however, so we offload some computation to the Segway PC, which we assume is trusted.

The robot uses a PointGrey Firefly camera [43] to detect visual landmarks. Landmarks are intended to correspond to prominent visual features in road environments, such as road signs, stop lines, and other easily and repeatably detectable objects. These landmarks are expected to be provided as part of the external map information, to enable map verification using sensor-reported landmark measurements. To simplify image-processing tasks in our experiments, we use ArUco tags [20] for our landmarks. OpenCV [39] is used to obtain the range and bearing of each tag in the camera's field of view, and this data is fed to the ZedBoard.

A Vicon motion-tracking system [57] provides robot-pose estimates and acts as a stand-in for an accurate satellite navigation system. A 2D SICK LIDAR [50] is used for detecting obstacles. A module on the Segway PC processes the raw LIDAR data to generate occupancy grids [54] for the ZedBoard.

The Segway PC also has a *path follower* for smoothing out the paths produced by the ZedBoard before sending wheel commands to the Segway. Our implementation is based on the pure pursuit algorithm [13].

4.3 Map data

An external map server provides the vehicle with a two-dimensional map of landmarks, lane positions, and other planning information. Each landmark location is a single point describing a sensor-visible feature in the environment. Lane-following information is provided as a *cost overlay* containing a planning cost for each grid cell in the map. The overlay penalizes travel outside of the lane boundaries with high planning costs, and is intended to be used in conjunction with the occupancy grid by the path planner.

Given the size of our experimental scenarios, it would be straightforward to save the entire map in memory on the robot. Autonomous road vehicles, on the other hand, operate in large-scale environments, and must continually download maps from an external source [34]. We model this by providing map information dynamically in segments covering the immediate area around the robot. The robot queries the map server for new segments when it has moved close to the edge of the current map segment.

A set of sparse map information is assumed to be present in memory of the vehicle at all times for verification. We call this information *secure sparse map data*, and is a much more scalable approach than saving all map data on-board.

4.4 Path planner

As an abstraction of a vehicle path planner, the Jif code on the ZedBoard implements an A* algorithm [22] to plan coarse paths, which are then given to the path follower on the Segway PC. Predefined *waypoints* along the lane are used sequentially as the planner goal. Both the cost overlay from the map and the occupancy grid of nearby obstacles contribute to the complete planning cost function. As the A* algorithm searches for the lowest-cost path, this cost function encodes a preference for lane following and obstacle avoidance in the planner.

4.5 Map verifier

We treat the external map server as potentially untrusted, so in order to use the map for planning, we require independent verification of the streaming detailed map information against local sensor data and the secure sparse map data. To achieve this, we assume a Gaussian measurement distribution around each expected landmark location, as reported by the map, and assess the agreement with local sensor measurements using a statistical hypothesis test.

First, we calculate the test statistic for a given sensor measurement $\boldsymbol{y} \in \mathbb{R}^2$ of a landmark, in map coordinates:

$$\Lambda = (\boldsymbol{y} - \boldsymbol{m}_0)^{\mathrm{T}} \boldsymbol{\Sigma}^{-1} (\boldsymbol{y} - \boldsymbol{m}_0)$$

where $\boldsymbol{m}_0$ is the map-reported landmark location. The covariance is assumed to be circular:

$$\boldsymbol{\Sigma} = \left(\sigma + \alpha \left\| \boldsymbol{m}_0 - \boldsymbol{x}^{\text{robot}} \right\| \right) \boldsymbol{I}$$

where $\boldsymbol{x}^{\text{robot}}$ is the current position of the robot, $\boldsymbol{I}$ is the 2×2 identity matrix, and α and σ are parameters that encode the constant and distance-dependent measurement noise, respectively. For a given set of sensor measurements, the hypothesis test is repeated for each map landmark in turn. If every hypothesis test rejects the null hypothesis (indicating a mismatch between expected and measured landmarks), then map verification fails, and the path planner does not use map information.

4.6 Information flow control

We implemented the planner and map verifier in 630 lines of Jif code.[2] Supporting this are 125 lines of Jif signatures (which allow

[2]All reported line counts omit comments and blank lines.

```
1  class Map[principal T, principal U] where T ≽ U {
2    OccuGrid{U←} overlay;
3    OccuGrid{T←} verifiedOverlay;
4  }
5
6  class Verifier {
7    void verify(Map[T,U]{T←} map,
8        ARTagMeasurement{T←}[]{T←} tags) {
9      if (verifyImpl(map, tags))
10       map.verifiedOverlay = endorseOverlay(map.overlay);
11     else map.verifiedOverlay = null;
12   }
13 }
14
15 class Planner {
16   Plan{T←} plan(Point{T←} start,
17       Point{T←} goal, OccuGrid{T←} grid,
18       Map[T,U]{T←} map) {
19     // If map is unverified, use contingency.
20     OccuGrid overlay = map.verifiedOverlay;
21     if (overlay == null)
22       return contingency(start, goal, grid);
23
24     // Combine overlay into occupancy grid and do A*.
25     grid.combine(overlay);
26     return astar(start, goal, grid);
27   }
28 }
```

Figure 5: Jif code sketch for verifier and planner.

Jif to interface with Java) and roughly 1000 lines of Java code for handling network communication and for Java to interface with Jif.

Figure 5 sketches three classes corresponding to the map, verifier, and planner. On line 1, the `Map` class is parameterized on two principals, `T` and `U`, which model trust levels in the system. The clause "`where T ≽ U`" means `T` is more trusted than `U`. Classes `Verifier` and `Planner` are also parameterized, but those annotations are elided for brevity.

Lane information is represented as a cost overlay, with low-cost areas representing the road. When downloaded from the map server, the map data is initially untrusted and stored in the `overlay` field (line 2). Once the map is verified, a copy of it is endorsed and stored in `verifiedOverlay` for use by the planner (lines 9–10).

If verification fails, then `verifiedOverlay` remains `null` (line 11), and the planner falls back on a contingency plan (line 22). Otherwise, when verification succeeds, the planner combines the map's cost overlay with the occupancy grid and runs an A* search on the result (lines 25–26).

The two key security types in this code are `OccuGrid{U←}` for `Map.overlay`, specifying that the overlay is untrusted, and `Plan{T←}` for the return value of `Planner.plan()`, specifying that the planner's output is trustworthy. Jif's type system statically ensures that the overlay is unable to influence planner output without the endorsement on line 10. The planner and map verifier use a total of 11 endorsements, summarized in Figure 6.

Location	Count	What is endorsed
`verifyImpl()`	2	Number of landmarks, verification result
`endorseOverlay()`	5	Overlay components: offset, size, resolution, cell contents, reference to overlay itself
Map constructor	2	Position & size of map
Map deserializer	2	Position & size of map

Figure 6: All endorsements in verifier and planner.

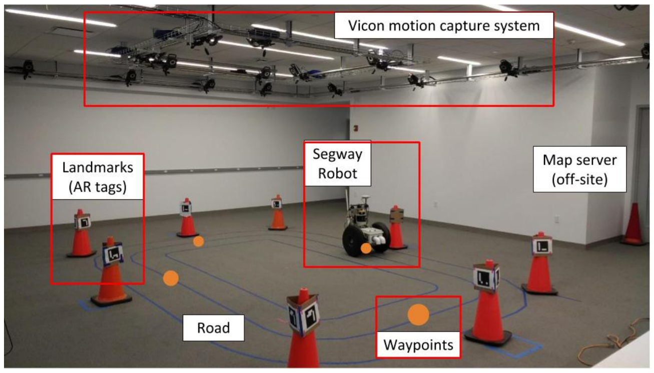

Figure 7: Experimental setup. Orange circles represent navigation waypoints provided in software.

4.7 Attack vector: malicious maps

The architecture is designed to mitigate potential attacks in which an untrusted hardware or software component is able to influence a critical driving function of the robot. This can arise when a software component is Internet-connected, or otherwise vulnerable to injection of malicious code, and is also connected to a critical driving software component. We assume that critical software components cannot be compromised directly, since they are running on a secure hardware/software stack, but their inputs may be manipulated if an untrusted source is attacked. In a modern automobile, such an attack might compromise the infotainment system and thereby gain access to steering, brake, or throttle controls via the vehicle CAN network, as was shown in [41].

The map server used in these experiments represents an external untrusted component. Future iterations of the prototype can have untrusted software running on the ZedBoard in tandem with the planner, to model an untrusted local software component running on the same processor as safety-critical computations. The co-location of untrusted and safety-critical software will make further hardware attacks possible, including timing-channel attacks [21] and other attacks exploiting shared hardware resources across tasks [24]. Implementing a secure verified processor on the ZedBoard FPGA can defend against these attacks.

4.8 Environment

Figure 7 shows the scenario we used to test the key features of the secure architecture. A simple loop lane is surrounded by 8 ArUco tag landmarks. Four trusted planning waypoints are provided ahead of time to the control stack, analogous to high-level navigation waypoints from a satellite navigation system. The waypoints are

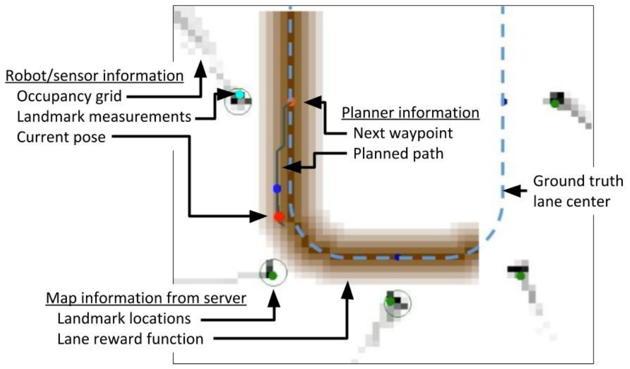

Figure 8: Visualization legend.

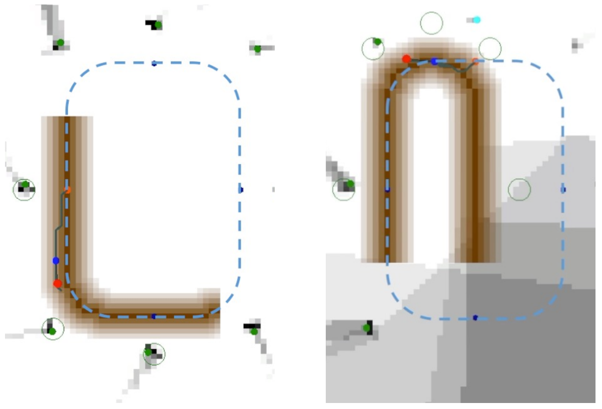
(a) Nominal map.
(b) Malicious map, with a deformation in the horizontal direction.

Figure 9: Segments from the externally provided map. The lane as reported by the map is shown in the brown gradient, and the true lane center is shown in dashed blue.

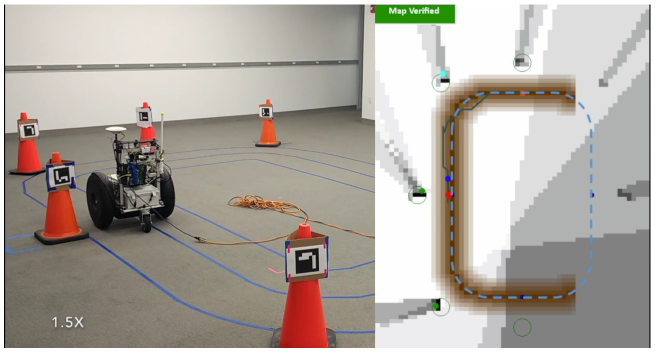

Figure 10: Case 1: The system has verified the map based on the agreement between the measured and expected landmark locations, shown in the upper-left corner of the visualization inlay.

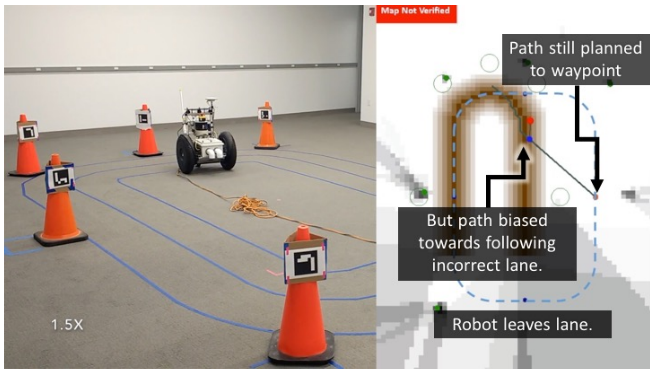
Map Not Verified
Path still planned to waypoint
But path biased towards following incorrect lane.
Robot leaves lane.
1.5X

Figure 11: Case 2: A planning failure as a result of the planner using incorrect map data, in the form of deformed lane and corresponding landmarks. The planner, operating on the malicious map, causes the vehicle to leave the lane.

provided to the planner in sequence so that the robot circles the loop clockwise. The sensor-only contingency plan, in the case of map verification failure, is to immediately decelerate the robot to a stop.

5 EVALUATION

5.1 Effectiveness of input validation

Figure 8 gives a visualization of the map and the robot's environment. The center of the actual lane geometry is shown as a dashed blue line. The robot's position appears as a red dot, and its current goal is shown as an orange dot. Green dots represent landmarks that were previously detected, and landmarks currently visible to the robot appear in bright cyan. The map server provides lane information as a cost overlay, shown in brown, and expected landmark positions, shown as thin green circles.

Figure 9 shows the nominal (a) and malicious (b) versions of the map. In the nominal map, the cost overlays and expected landmark positions correctly describe the true lane geometry and landmarks. The malicious map is deformed such that the driving loop is narrower than in the nominal case. The landmark locations in the malicious map are similarly transformed, so that they no longer match physical landmarks.

We experimentally evaluated the autonomous CPS prototype for three cases: 1) the map server provides a correct map of lane geometry and landmarks (Figure 9(a)) and the secure Jif-based planner is used; 2) the map server provides the malicious map shown in Figure 9(b), while the robot uses an insecure Java version of the planner; and 3) the map server provides the same malicious map, and the secure Jif-based planner is used. A software visualization (Figure 8) presents the measurement and map data used in map verification and other decision making. Video results of all three cases are available online [31].

A still image from Case 1 is shown in Figure 10. In this test, the map was successfully verified, since actual landmark measurements corresponded to their expected locations on the map. The planner executed as expected. The robot successfully navigated the loop.

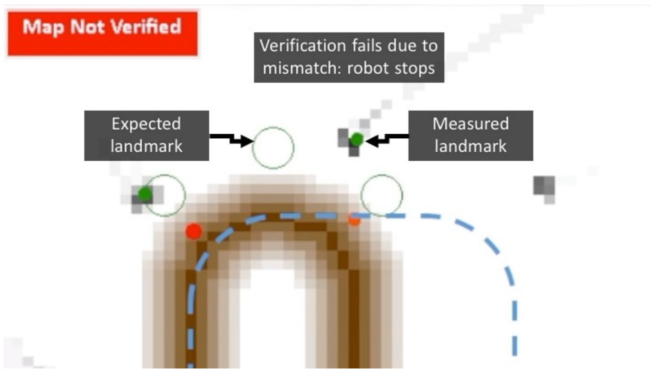

Figure 12: Case 3: A correct planning response resulting from map verification failure.

In Case 2, the insecure, Java-based planner contains a bug that causes the map to be always used, even when verification fails.[3] Initially, the lane geometry of the deformed map is sufficiently similar to the true lane geometry, so the robot maintains position in the lane. However, as the robot rounds the curve at the top of the map, the landmark measurements begin to diverge from the expected landmark locations. This causes map verification to fail, but since the planner does not consider the map trust level, a plan is generated that follows the map-provided lane geometry. This causes the vehicle to leave the physical lane, shown in Figure 11.

In Case 3, the secure, Jif-based planner is served the malicious map. As in Case 2, the landmarks on the deformed map are initially consistent with sensor observations, so the map is verified and driving continues nominally. However, when verification fails at the top of the course, the map is not endorsed for use in the secure planner. The planner therefore generates a safe and secure, sensor-only plan, not reliant on map data. This causes the vehicle to slowly come to a full stop without leaving the lane, ensuring a safe stoppage without relying on corrupt map data.

5.2 Overhead of software-level information flow control

We evaluated programming overhead by comparing our Jif-based verifier and planner against the insecure Java version, which was written in roughly 1,300 lines of code.

The secure version contains 630 lines of Jif code, which roughly correspond to 537 lines of the Java version, and involve changing (or adding) 351 lines of code. Of these changed lines, 152 involve adding security annotations (including 11 endorsements), and 30 lines were changed to copy the map as it is verified and endorsed. The remaining 169 lines were miscellaneous changes, such as introducing the security bug, switching to Jif's data structure libraries (which track information flows), and refactoring network communication elsewhere. This overhead seems reasonable, and Figure 5 gives a representative sample of Jif code: nearly all annotations appear on class fields and method headers.

[3]The analogous change to Figure 5 would be to replace `map.verifiedOverlay` with `map.overlay` on line 20, although the Jif compiler would reject this change as being insecure.

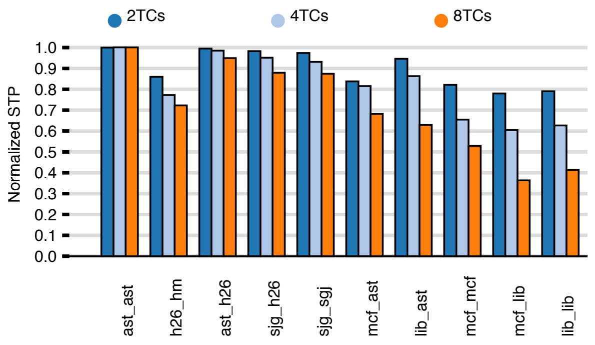

Figure 13: Norm. STP as the number of TCs increases.

The run-time performance of Jif is similar to that of Java. Most label annotations are erased by the compiler, so do not impose a run-time overhead. Any security checks that occur at run time are checks that the programmer is forced to write; omitting them would cause the program to be insecure.

Endorsement is the largest source of overhead, since data must be copied to be endorsed. Although our verifier endorses the copied map, the overhead is not significant, since endorsement is only done periodically, when the map changes. Indeed, the robot exhibited no noticeable performance difference between the two versions of the planner.

5.3 Overhead of timing-interference protection

The baseline RISC-V Rocket processor uses 34,508 (64.9%) of the look-up tables (LUTs) on the ZedBoard's 7z020clg484-1 FPGA. By comparison, our processor with strong information flow control uses 40,205 LUTs (75.6%), a LUT utilization overhead of 16.5%. The baseline processor uses 13 (9%) of the block RAM tiles, whereas the secure processor utilizes 19.5 (14%). Most of this area overhead is due to the security tags being stored on-chip.

The baseline and the secure processor both have the same clock frequency. The main performance overhead of the secure processor came from a constant-time multiplier that we introduce to remove timing interference. The average performance overhead for our processor prototype was 12.4%. This overhead represents the cost for a simple embedded processor with conservative protection schemes.

To study the performance overhead of strong timing isolation for modern multi-core processors, we simulated the multicore processor [8] integrated with DRAMSim2 [47], modeling today's out-of-order multi-core processor with memory hierarchy parameters from Intel Xeon E3-1220L and Intel Xeon E7-4820. The performance is evaluated for multiprogram workloads comprised of SPEC2006 benchmarks, which is the standard benchmark for processor performance. The system throughput (STP) is used as a metric. STP is the aggregated normalized IPC (instructions per cycle) of each program relative to the IPC when that program runs by itself.

Figure 13 shows performance of timing compartments (TCs) as the number of TCs and cores increases from 2 to 8. Overhead is less than 5% for compute-intensive benchmarks, even with a large number of TCs. Yet, the overhead of memory-intensive workloads increases with the number of TCs, because more compartments share the same amount of fixed memory bandwidth. However, for secure CPS, two timing compartments should suffice to isolate safety-critical software. In this case, simulation results suggest that performance overhead is less than 20% even for memory-intensive operations.

6 RELATED WORK

The community has looked at security issues in cyber-physical systems from many angles. In this work, we tackle the problem by integrating secure control algorithms, software-level information flow security, and secure hardware. We summarize related work in control-algorithm security, software-level formal methods, verified hardware, as well as full system integration.

6.1 Robotic attack modalities

Several adversarial attack vectors have recently been studied. These attack vectors can roughly be decomposed into attacks on robotic control inputs and on robotic sensors. Checkoway et al. [11] perform an analysis of attacks against a conventional vehicle. They categorize the attacks into indirect physical access, and short- and long-range wireless access. They show that all attack modalities enable full control of the vehicle's CAN bus and all Electronic Control Units (ECUs). These exploits fall into the category of a control-input attack (braking, throttle, traction control, etc).

Recent studies have also dealt with attacks on robotic sensors, such as [32, 40]. These works analyze the effects of compromising the sensing data used to inform the robot about its position in the world and the state of its environment, and give a real-world example of the consequences of such an attack. It is well known that the state-of-the-art autonomous vehicles rely heavily on detailed map data [7], and the possibility of map-poisoning attacks has been identified [41], but to the authors' knowledge, no prior work has analyzed a method for adequately addressing these attacks.

Security issues with inter- and intra-vehicle networks has been addressed with research in the VANET (vehicle ad hoc network) and MANET (mobile ad hoc network) communities [12, 66]. This research typically focuses on addressing security before the data gets to the car via the communication protocols and verification via many inputs, whereas the proposed work here focuses on a dedicated data feed for map data, and on-board sensors. We assume that the communications could be compromised, and thus ask the question: what can we do to secure the vehicle.

6.2 Security in control algorithms

As part of the DARPA HACMS project, Pajic et al. [40] studied the theory (and validation) of multiple signals considered here. The previous verification methods were developed for linear systems with bounded inputs, showing that guarantees could be made: given $\frac{N}{2} + 1$ trusted inputs, the other untrusted inputs could be detected. Extensions have focused on time histories of these signals. Our work here focuses on general signals with no assumptions on dynamics or distributions. Similarly, anomaly node detection has also been studied in vehicular [55] and wireless[60] ad hoc sensor networks. Usually a compromised node is discovered by its own statistics [14] or interaction with other nodes [33]. This has to be checked for all the nodes in the networks. Our work, on the other hand, uses information flow theory to propagate the trustworthiness among sensor inputs, and only needs to check sensors labeled as untrusted.

Sha et al. [49] propose a complementary method that wraps a complex control algorithm with a simple, secure control algorithm with only trusted inputs to ensure CPS safety and reliability. Our work follows this architecture in the sense that a complex map-based path planner is gated by a simple sensor-only planner in our system. Rather than rely on manual code auditing, we use automatic verification of information flow for security assurance.

6.3 Formal methods for CPS security

We use Jif, a programming language that enforces software-level information flow security with its type system. Information flow methods have not been applied much to cyber-physical systems; however, one related prior effort is by Morris et al. [36], who study the vulnerability of cyber-physical systems to compromised inputs from the perspective of quantitative information flow.

Other formal methods have also been adopted to ensure security in CPS. A heavier-weight approach is to use a proof assistant to certify functional correctness. For example, control software of a quadcopter has been verified using the Coq proof assistant [53], ensuring that it stays away from restricted airspace [45]. ROSCoq [3] integrates Coq into ROS [46], a popular framework for robot control. This integration helps software designers to formally verify ROS implementations and ensure safety. Timing verification of CPS is also essential to physical security. Ziegenbein and Hamann [68] verify the timing of safety-critical automotive software by using SpaceEx [19] to systematically enumerate all timing conditions. Formal analysis methods have also been used to verify the timeliness of real-time control systems [18].

Previous efforts use formal verification tools with a steep learning curve to certify security properties. By contrast, this work is the first, to our knowledge, that adopts lightweight, programming language-based methods to provide security guarantees in CPS.

6.4 Secure HDL and verified processor

Secure HDL has recently been proposed as a way to verify information flow security of hardware [28, 29, 67]. Caisson [29] is an HDL that supports purely static labels in information flow tracking. SecVerilog [67] adopts static information flow with mutable, dependent security labels on top of Verilog to enforce policy and eliminate timing channels. Sapper [28], instead of static verification, adds information flow tracking logic into the original hardware. Multiple verified processors [16, 56] have been constructed using secure HDL. This work is also inline with this methodology. While the tool itself is not new, this work represents the first to show that a secure multi-core processor with no timing interference can be designed and verified. This work is also the first to evaluate the performance overhead of such strong timing isolation.

6.5 Secure CPS integration

This work aims to vertically integrate different protection mechanisms to provide security guarantees in CPS. Parallel to our effort, Veriphy [10] integrates a self-certified toolchain that generates trusted assembly code from a high-level specification of a control algorithm in a CPS design. The high-level control algorithm specification is also mathematically verified by a logic calculus. However, this work lacks hardware verification.

As a complementary approach to hardening CPS against attack, a separate line of work has explored fast recovery from vulnerabilities. Restart-based security [1, 2, 4] aims to guarantee the security of CPS by restarting the full system. In the threat model, it is assumed that it takes non-zero time for an adversary to take control and cause physical damage. The system can thus nullify attacks by proactively restarting to remove the adversary before it has any effect. This is suitable for applications that tolerate availability loss during restart. Our integration, on the other hand, provides higher availability because it has no offline period.

7 CONCLUSION AND FUTURE WORK

Our contribution is a novel system architecture for secure autonomous cyber-physical systems, with key components including verified hardware, language-based information-flow control in software, and online algorithms for input verification. This vertical integration of security offers a desirable method to ensure secure operation of cyberphysical systems.

A proof-of-concept demonstration was developed using an indoor mobile robotic testbed performing a lane-following task. Experimental results show that the map-verification strategy successfully prevents adverse vehicle behavior in the event of a maliciously compromised map. By implementing the planning and verification software in the Jif language, separation is guaranteed between trusted and untrusted information flows. Additionally, using Jif guides programmers to consider security when developing software—during the development of the secure planner, a number of bugs were discovered where the compiler prevented the use of unverified, untrusted information.

While this study demonstrates the importance and the promise of the vertically-integrated protection approach in building secure CPS, our experiences with the prototype also show limitations of protecting each layer separately and suggest that there is a potential to further improve the security and the efficiency of the integrated protection approach through tighter integration across system layers. For example, the traditional information flow control techniques in both hardware and software require a system to make discrete decisions about the security level of information, such as determining whether input is either trusted or untrusted. A more continuous notion of trust in information flow control will allow a system to more accurately reason about the trustworthiness of inputs. In hardware, today's mechanisms for strong software isolation need to remove timing interference among critical software modules even when control algorithms can tolerate certain delays. A tighter integration between control algorithms and hardware isolation has a potential to provide strong security with lower overhead by selectively removing interference when needed.

Future work will focus on extending the prototype system to fully integrate all components of the protection architecture including custom hardware, develop more tightly-integrated protection techniques, and to demonstrate protection against more threat vectors. For example, in this study, we evaluated the overhead of secure hardware using simulations and an RTL prototype. We are in the process of developing secure processor hardware capable of running a full operating system. This processor will be integrated into the prototype robot in order to study the fully integrated system.

ACKNOWLEDGMENTS

This research has been supported by NSF grant 1513797 and NASA grant NNX16AB09G. Any opinions, findings, conclusions, or recommendations here are those of the authors and do not necessarily reflect the views of any funding agencies.

REFERENCES

[1] F. A. T. Abad, R. Mancuso, S. Bak, O. Dantsker, and M. Caccamo. Reset-based recovery for real-time cyber-physical systems with temporal safety constraints. In *Emerging Technologies and Factory Automation (ETFA), 2016 IEEE 21st International Conference on*, pages 1–8. IEEE, 2016.
[2] F. Abdi, R. Tabish, M. Rungger, M. Zamani, and M. Caccamo. Application and system-level software fault tolerance through full system restarts. In *Proceedings of the 8th International Conference on Cyber-Physical Systems*, pages 197–206. ACM, 2017.
[3] A. Anand and R. Knepper. Roscoq: Robots powered by constructive reals. In *International Conference on Interactive Theorem Proving*, pages 34–50. Springer, 2015.
[4] M. Arroyo, H. Kobayashi, S. Sethumadhavan, and J. Yang. Fired: Frequent inertial resets with diversification for emerging commodity cyber-physical systems. *arXiv preprint arXiv:1702.06595*, 2017.
[5] K. Asanović, R. Avižienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, P. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman. The Rocket Chip generator. Technical Report UCB/EECS-2016-17, University of California Berkeley, Apr. 2016.
[6] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security.* Addison Wesley, Apr. 2003. ISBN 0321136160.
[7] B. Berman. Whoever Owns the Maps Owns the Future of Self-Driving Cars, 2016.
[8] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, Aug. 2011.
[9] R. Bishop. *Intelligent Vehicle Technology and Trends.* 2005.
[10] B. Bohrer, Y. K. Tan, S. Mitsch, M. O. Myreen, and A. Platzer. Veriphy: verified controller executables from verified cyber-physical system models. In *Proc. 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 617–630. ACM, 2018.
[11] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, T. Kohno, et al. Comprehensive experimental analyses of automotive attack surfaces. In *USENIX Security Symposium*. San Francisco, 2011.
[12] T. W. Chim, S. M. Yiu, L. C. K. Hui, and V. O. Li. Vspn: Vanet-based secure and privacy-preserving navigation. *IEEE Transactions on Computers*, 63(2):510–524, Feb 2014.
[13] R. C. Coulter. Implementation of the pure pursuit path tracking algorithm. Technical report, Carnegie-Mellon University, 1992.
[14] D.-I. Curiac, O. Banias, F. Dragan, C. Volosencu, and O. Dranga. Malicious node detection in wireless sensor networks using an autoregression technique. In *Networking and Services, 2007. ICNS. Third International Conference on*, pages 83–83. IEEE, 2007.
[15] R. Descartes. *Meditations on First Philosophy.* 1641.
[16] A. Ferraiuolo, R. Xu, D. Zhang, A. C. Myers, and G. E. Suh. Verification of a practical hardware security architecture through static information flow analysis. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.
[17] A. Ferraiuolo, Y. Zhao, A. C. Myers, and G. E. Suh. HyperFlow: A processor architecture for nonmalleable, timing-safe information flow security. In *Proceedings of the 25th ACM Conference on Computer and Communications Security*, Oct. 2018.
[18] G. Frehse, A. Hamann, S. Quinton, and M. Woehrle. Formal analysis of timing effects on closed-loop properties of control software. In *Real-Time Systems Symposium (RTSS), 2014 IEEE*, pages 53–62. IEEE, 2014.
[19] G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler. SpaceEx: Scalable verification of hybrid systems. In *International Conference on Computer Aided Verification*, pages 379–395. Springer, 2011.
[20] S. Garrido-Jurado, R. M. noz Salinas, F. Madrid-Cuevas, and M. Marín-Jiménez. Automatic generation and detection of highly reliable fiducial markers under

occlusion. *Pattern Recognition*, 47(6):2280 – 2292, 2014.

[21] Q. Ge, Y. Yarom, D. Cock, and G. Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, pages 1–27, 2016.

[22] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[23] M. Hicks, C. Sturton, S. T. King, and J. M. Smith. SPECS: A Lightweight Runtime Mechanism for Protecting Software from Security-Critical Processor Bugs. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[24] G. Irazoqui, T. Eisenbarth, and B. Sunar. Cross processor cache attacks. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 353–364. ACM, 2016.

[25] T. Jeske. Floating car data from smartphones: What Google and Waze know about you and how hackers can control traffic. In *Proc. BlackHat Europe*, pages 1–12, 2013.

[26] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proc. ACM 22nd Symp. on Operating System Principles (SOSP)*, pages 207–220, 2009.

[27] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints*, Jan. 2018.

[28] X. Li, V. Kashyap, J. K. Oberg, M. Tiwari, V. R. Rajarathinam, R. Kastner, T. Sherwood, B. Hardekopf, and F. T. Chong. Sapper: A Language for Hardware-level Security Policy Enforcement. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[29] X. Li, M. Tiwari, J. Oberg, F. T. Chong, T. Sherwood, and B. Hardekopf. Caisson: A hardware description language for secure information flow. In *Proceedings of the 32nd ACM Conference on Programming Language Design and Implementation*, San Jose, California, USA, June 2011.

[30] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown. *ArXiv e-prints*, Jan. 2018.

[31] J. Liu, J. Corbett-Davies, A. Ferraiuolo, M. Campbell, G. E. Suh, and A. C. Myers. Videos of demo of self-driving robot with map verification. Online, http://hdl.handle.net/1813/52638, Oct. 2017.

[32] D. Majumdar. Iran Claims Successful Test Flight of Stealth UAV, 2014.

[33] M. Mathews, M. Song, S. Shetty, and R. McKenzie. Detecting compromised nodes in wireless sensor networks. In *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2007. SNPD 2007. Eighth ACIS International Conference on*, volume 1, pages 273–278. IEEE, 2007.

[34] R. McMillan. Siemens: Stuxnet worm hit industrial systems, 2010.

[35] C. Miller and C. Valasek. Remote exploitation of an unaltered passenger vehicle. 2015.

[36] E. R. Morris, C. G. Murguia, and M. Ochoa. Design-time quantification of integrity in cyber-physical systems. In *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 63–74, 2017.

[37] T. Moscibroda and O. Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, 2007.

[38] A. C. Myers. JFlow: Practical mostly-static information flow control. In *26^{th} ACM Symp. on Principles of Programming Languages (POPL)*, pages 228–241, Jan. 1999.

[39] OpenCV. `www.opencv.org`.

[40] M. Pajic, I. Lee, and G. J. Pappas. Attack-resilient state estimation for noisy dynamical systems. *IEEE Transactions on Control of Network Systems*, 4(1):82–92, 2017.

[41] J. Petit and S. E. Shladover. Potential cyberattacks on automated vehicles. *IEEE Transactions on Intelligent Transportation Systems*, 16(2):546–556, 2015.

[42] J. Petit, B. Stottelar, M. Feiri, and F. Kargi. Remote attacks on automated vehicles sensors: Experiments on camera and LiDAR. In *Black Hat Europe*, 2015.

[43] PointGrey. `www.ptgrey.com`.

[44] S. Poslad. *Ubiquitous computing: smart devices, environments and interactions.* John Wiley & Sons, 2011.

[45] D. Ricketts, G. Malecha, and S. Lerner. Modular deductive verification of sampled-data systems. In *Proceedings of the 13th International Conference on Embedded Software*, page 17. ACM, 2016.

[46] Robot Operating System. `www.ros.org`.

[47] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. DRAMSim2: A cycle accurate memory system simulator. *Computer Architecture Letters*, 2011.

[48] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on selected areas in communications*, 21(1):5–19, 2003.

[49] L. Sha. Using simplicity to control complexity. *IEEE Software*, 18(4):20–28, 2001.

[50] SICK, Inc. `www.sick.com`.

[51] J. Singh, T. Pasquier, J. Bacon, H. Ko, and D. Eyers. Twenty security considerations for cloud-supported Internet of Things. *IEEE Internet of Things Journal*, 3(3):269–284, 2016.

[52] S. Skorobogatov and C. Woods. Breakthrough silicon scanning discovers backdoor in military chip. In *Cryptographic Hardware and Embedded Systems Workshop*, September 2012.

[53] The Coq Development Team. The Coq proof assistant, version 8.8.0, Apr. 2018.

[54] S. Thrun, W. Burgard, and D. Fox. *Probabilistic robotics.* 2005.

[55] D. Tian, Y. Wang, G. Lu, and G. Yu. A vehicular ad hoc networks intrusion detection system based on busnet. In *Future Computer and Communication (ICFCC), 2010 2nd International Conference on*, volume 1, pages V1–225. IEEE, 2010.

[56] M. Tiwari, J. Oberg, X. Li, J. K. Valamehr, T. Levin, B. Hardekopf, R. Kastner, F. T. Chong, , and T. Sherwood. Crafting a usable microkernel, processor, and I/O system with strict and provable information flow security. In *ISCA'11*, June 2011.

[57] Vicon Motion Systems, Ltd. `www.vicon.com`.

[58] R. Wojtczuk and J. Rutkowska. Attacking SMM Memory via Intel CPU Cache Poisoning. invisiblethingslab.com/resources/misc09/smm_cache_fun.pdf, 2009.

[59] R. Wojtczuk and J. Rutkowska. Following the White Rabbit: Software Attacks Against Intel VT-d Technology. http://theinvisiblethings.blogspot.com/2011/05/following-white-rabbit-software-attacks.html, 2011.

[60] M. Xie, S. Han, B. Tian, and S. Parvin. Anomaly detection in wireless sensor networks: A survey. *Journal of Network and Computer Applications*, 34(4):1302–1325, 2011.

[61] T. K. Yaakov Bar-Shalom, Xiao-Rong Li. *Estimation with Applications to Tracking and Navigation.* Wiley, New York, 1st edition, 2001.

[62] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Secure program partitioning. *ACM Trans. on Computer Systems*, 20(3):283–328, Aug. 2002.

[63] ZedBoard. `www.zedboard.org`.

[64] K. C. Zeng, Y. Shu, S. Liu, Y. Dou, and Y. Yang. A practical gps location spoofing attack in road navigation scenario. In *Proceedings of the 18th International Workshop on Mobile Computing Systems and Applications*, pages 85–90, 2017.

[65] K. Zetter. Headline: Hackers could commandeer new planes through passenger wi-fi, April 2015.

[66] C. Zhang, R. Lu, X. Lin, P.-H. Ho, and X. Shen. An efficient identity-based batch verification scheme for vehicular sensor networks. In *IEEE INFOCOM 2008 - The 27th Conference on Computer Communications*, pages 246–250, April 2008.

[67] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers. A Hardware Design Language for Timing-Sensitive Information-Flow Security. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[68] D. Ziegenbein and A. Hamann. Timing-aware control software design for automotive systems. In *Proceedings of the 52nd Annual Design Automation Conference*, page 56. ACM, 2015.